MOORE, GENE WALL. Two-Dimensional Grid Grammars. (1977)
Directed by: Dr. Richard Michael Willett. Pp. 70.

The purpose of this thesis is to introduce an original approach to two-dimensional grammars. A class of machines which operate on two-dimensional tapes called grids is defined. Grid machines operate in a manner similar to Turing machines but have the ability to move over the grid in four directions. The concept of grammars is extended to two-dimensional grids. A language generated by a grid grammar consists of a set of pictures. The production rules of a grid grammar focus on certain regions of a grid and ignore the remaining regions. By imposing a set of increasingly severe restrictions on the non-ignored regions, a hierarchy of grid grammars is obtained. Descriptions and illustrations of Turing machines, the Wang programming language, and one-dimensional grammars are included.

TWO-DIMENSIONAL GRID GRAMMARS

by

Gene Wall Moore

A Thesis Submitted to
the Faculty of the Graduate School at
The University of North Carolina at Greensboro
in Partial Fulfillment
of the Requirements for the Degree
Master of Arts

Greensboro
1977

Approved by

Thesis Adviser

APPROVAL PAGE

This thesis has been approved by the following committee of the Faculty of the Graduate School at the University of North Carolina at Greensboro.

Thesis Adviser Michael Willett

Committee Members Michael Willett
William A Bowers
E.E. Posey

December 19, 1977
Date of Acceptance by Committee

ACKNOWLEDGMENTS

The writer wishes to express her appreciation to Dr. Michael Willett for his guidance and patience throughout this study.

Appreciation is extended to Dr. E. E. Posey and Dr. William A. Powers for their cooperation as members of the examining committee.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (continued)

CHAPTER I

INTRODUCTION

Everyone has an intuitive understanding of the concept of an algorithm. It is generally agreed that an algorithm, or effective procedure, is a finite set of instructions which meet certain requirements. All instructions must be unambiguous and must not involve any element of chance. Each instruction must be executed in a finite amount of time. It is usually not required that algorithms halt. Instead, the set of instructions must be so specifically stated that any two people executing them would perform precisely the same operations.

In this thesis we first introduce a class of mathematical machines which are used to define the concept of an algorithmic computation. These Turing machines [3], which operate on one-dimensional tapes, are defined in Chapter II. We consider machines as computational devices, function evaluators,and acceptor automata for syntactically correct imput. A set of instructions, called the Wang programming language, which can be assembled in such a way as to simulate a given Turing machine, is presented in Chapter III.

In Chapter IV we define grammars, which are systems for generating strings of symbols with certain structured properties. The Chomsky Hierarchy of classes of restricted grammars is outlined [1].

In Chapter V we introduce a class of machines which operate in a manner similar to Turing machines but on a two-dimensional tape. We refer to this class of machines as grid machines.

In the literature [2,4] there is no standard approach to extending the concept of grammars to higher dimensions. In this thesis we propose an original approach to this problem and examine some of its ramifications. We extend the notion of grammars to two-dimensional grids and define grid grammars which generate sets of pictures.

In Chapter VI we present a classification of restricted grid grammars.

Chapter VII is a summary of the original concepts of grid machines and grid grammars presented in this thesis.

CHAPTER II

TURING MACHINES

At the turn of the century the formalistic approach to mathematics was in full swing under the leadership of David Hilbert. The formalists sought to develop formal axiomatic systems for mathematics. An axiomatic system consists of a set of undefined terms called primitives and a set of statements about the primitives called axioms, which one assumes to be true. In this way the axioms serve to partially restrict and interrelate the primitive terms. Any statement that can be derived from a finite string of logical inferences beginning with the axioms is called a theorem of the system. If no contradictory theorems can be derived, then the axioms are called consistent. A model of an axiomatic system is any example of the system in which the primitive terms are given particular meaning. A statement is called logically true if it is true in every model of the system when the terms in the statement are properly interpreted in each model. One can see that each theorem of a system must be logically true.

The formalists were concerned not only with the content of the axioms but even more with the way axioms are formed, how proofs are structured, how models of the axioms are constructed, and with other questions considered metamathematical in nature. The ultimate goal was to construct an axiomatic system which contained no inherent contradictions (a consistent system) and which contained abstract versions of all

of present day mathematics; that is, set theory, number theory, logic, etc. The formalists wanted to develop a consistent axiomatic system complete enough to the extent that all logically true statements would be theorems. They also hoped to develop general procedures for finding proofs of these theorems.

However, in 1931 Kurt Gödel published two theorems which resulted in the destruction of most of Hilbert's basic goals. One theorem states that if any consistent system contains an abstraction of the natural numbers (i.e., induction), then not all logically true statements are theorems of the system. Gödel's other theorem stated that any system such as described in his first theorem necessarily will not contain procedures for checking its own consistency.

Godel's first theorem indicates that there will never be a general procedure for determining whether or not a given statement in mathematics is logically true. There will always exist true statements which cannot be proved. This result caused the formalists to shift their investigations to better defining the process of proof itself. Among the questions that confronted them were: Can one develop a theory of automated or mechanical theorem proving? Can the notions of effective procedure and algorithm be formally and satisfactorily defined? We shall restrict ourselves to procedures which involve the manipulation of symbols and strings of symbols.

In 1936 Alan M. Turing, a logician, presented his version of a definition for the intuitive notion of effective procedure. He described a class of mechanistic procedures, called Turing machines, which perform

algorithmic computations. Turing believed that "every effective procedure can be performed by some Turing machine." This thesis (in quotes) is generally accepted, although it cannot be proved because the term effective procedure would need a definition.

Quite different approaches to the idea of effective procedure have been advanced by Kurt Godel, S.C. Kleene, Emil Post, and later Andrei Markov and Raymond Smullyan. These systems have all turned out to be equivalent to Turing machines. All of these systems define exactly the same class of procedures. This fact gives power and credibility to Turing's thesis.

A Turing machine is a model of a computational process which accepts strings of symbols as input and produces output according to an algorithm. The output of a Turing machine is determined by the input and by the structure of the machine. We can also interpret this computation as function evaluation with the input corresponding to the function argument and the output representing the function value. In this chapter we will first describe the structure and behavior of a Turing machine, then establish the correspondence between machine computation and function evaluation, and finally consider the ability of a Turing machine to recognize syntactically correct input.

A Turing machine differs from what is usually termed a finite state machine by its ability to re-examine squares on its input tape.

A Turing machine consists of three parts as pictured schematically in Figure 2.1:

(a) a control unit
(b) a read/write head
(c) an unbounded tape used for input and output

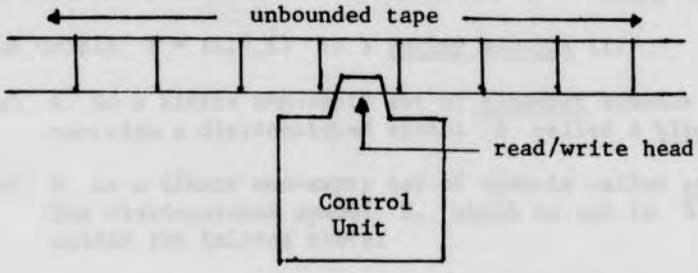


Figure 2.1 Turing machine

The tape consists of a doubly-infinite row of squares, all but a finite number of which are initially blank. The marked squares may contain any symbol from a given finite alphabet. At any moment in time the control unit scans one square of the tape through the read/write head.

The Turing machine operates sequentially on the tape as follows:

(a) The read/write head is placed over a particular starting square, and the control unit is placed in an initial state.

(b) After reading the symbol printed in the scanned square, the control unit writes some alphabet symbol on the square which is a function of the state of the control unit and the input symbol.

(c) The initial state of the control unit is changed.

(d) The read/write head is moved one square to the right or left.

These actions constitute one time unit of computation.

If during the computation the machine enters a specially designated halting state, computation ends, and the result of the computation corresponds to the final sequence of symbols appearing on the tape. If the machine never enters the halting state, computation continues indefinitely.

We will now present a formal definition of a Turing machine.

The triple $T = (A,S,f)$ is a Turing machine if:

(a) A is a finite non-empty set of alphabet symbols which contains a distinguished symbol B called a blank.

(b) S is a finite non-empty set of symbols called states. The distinguished symbol H, which is not in S, is called the halting state.

(c) f is a function such that $f:A \times S \to A \times (S \cup \{H\}) \times \{R,L,N\}$ with the property that $\delta' = H$ if and only if $M = N$ in $f(a,\delta) = (b,\delta',M)$. We shall see later that this restriction on the computing function f simply requires that the read/write head does not move after the control unit has entered the halting state.

The computing function provides the output symbol, the next state, and the read/write head movement as a function of present state and present input as required above. Since there are only a finite number of function arguments, the computing function is usually presented in tabular or graphic form.

The tabular form consists of a set of 5-tuples:

a	δ	b	δ'	M

where a is the input symbol,
δ is the present state symbol,
b is the output symbol,
δ' is the next state symbol,
M is the symbol indicating direction of read/write head movement.

We will usually represent each Turing machine graphically by converting each 5-tuple, $a\ \delta\ b\ \delta'\ M$, in the function table into an arrow connecting two state circles as in Figure 2.2.

Consider the Turing machine M_1 with $A = \{0,1,B\}$ and $S = \{1,2,3,4,5\}$ whose computing function is given in Table 2.3.

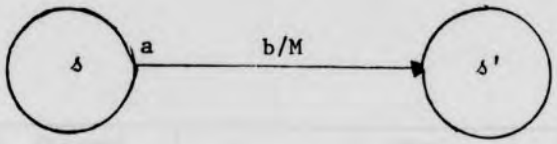


Figure 2.2 State Diagram of the Computing Function

The graphical form of M_1 is given in Figure 2.4. The arrow pointing to state 1 indicates that this state has been selected as the initial state of the machine. Multiple labels on an arrow will be used when possible in order to simplify the graphical form of a Turing machine.

The operation of M_1 on the input tape is diagrammed for each time unit in Figure 2.5. At time unit $t = 0$, M_1 is in the initial state, scanning the leftmost, nonblank symbol. The present state (δ) of the machine is displayed in the control unit symbol. M_1 reads the input symbol, moves left one square, and prints that symbol. M_1 then moves 2 squares right and repeats the above operations. The reader will notice that this program moves the finite number of input symbols one square left on the tape.

Unless otherwise stated for a specific machine, we will hereafter denote the alphabet by $A = \{a_1, a_2, \ldots, a_n\}$. A^* will denote the set of all finite strings (tapes) of symbols from A, repetitions allowed, with ϕ used to designate the empty string. A^+ is A^* minus the empty string. When reference is made to a lexigraphical ordering of tapes, tapes will be listed in increasing length, and for a given length

Input	Present State	Output	Next State	Read/write Head Movement
0	1	0	4	L
1	1	1	2	L
B	1	B	5	L
0	2	1	3	R
1	2	1	3	R
B	2	1	3	R
0	3	0	1	R
1	3	1	1	R
B	3	B	1	R
0	4	0	3	R
1	4	0	3	R
B	4	0	3	R
0	5	B	H	N
1	5	B	H	N
B	5	B	H	N

Table 2.3 Computing Function Table of M_1

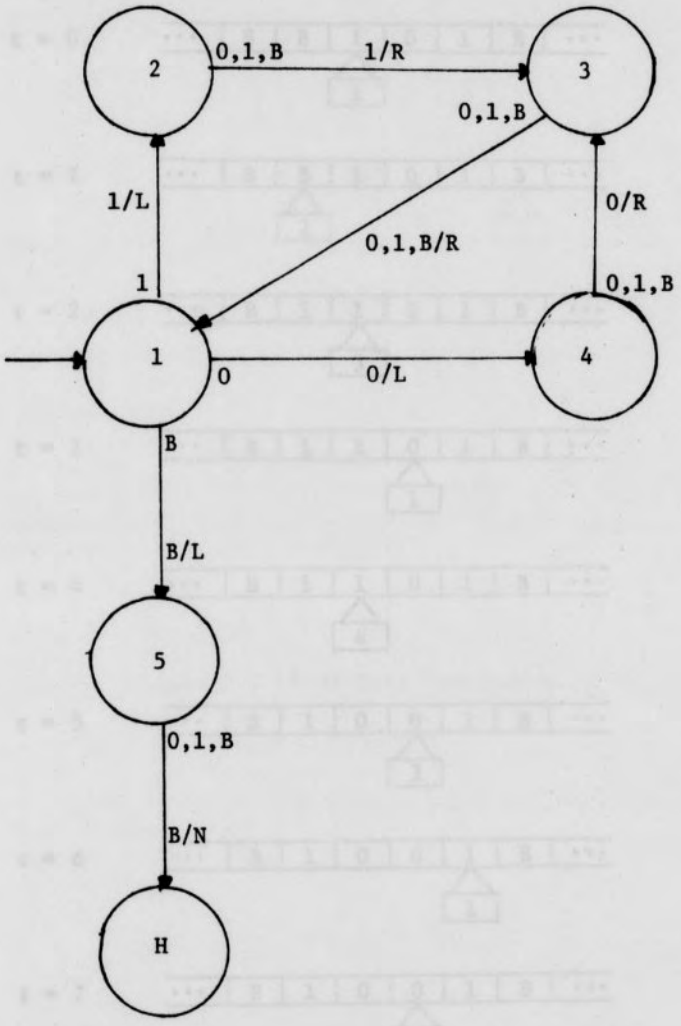


Figure 2.4 State Diagram of M_1

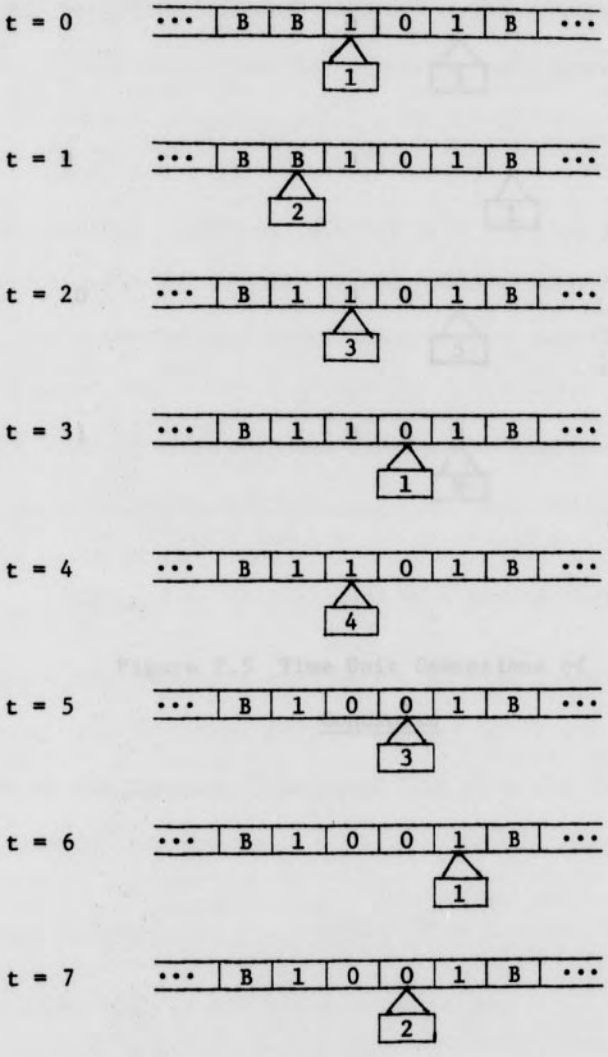


Figure 2.5 Time Unit Operations of M_1

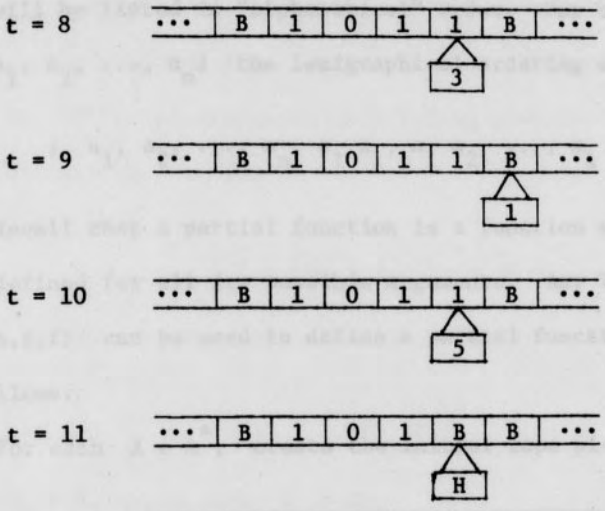


Figure 2.5 Time Unit Operations of M_1

Continued

they will be listed in "alphabetical" order. For example, with $A = \{a_1, a_2, \ldots, a_n\}$ the lexigraphical ordering would be

$$\phi, a_1, a_2, \ldots, a_n, a_1\,a_1, a_1\,a_2, \ldots, a_1\,a_n \ldots.$$

Recall that a partial function is a function which is not necessarily defined for all its possible arguments. Any Turing machine $T = (A,S,f)$ can be used to define a partial function from A^* to A^* as follows:

For each $X \in A^*$, create the initial tape pictured in Figure 2.6

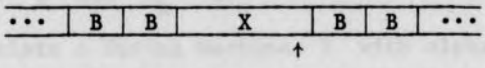


Figure 2.6 Initial Tape of a Turing Machine

Start the Turing machine in its initial state on the rightmost symbol of X. If the machine eventually halts and $Y \in A^*$ is the segment of the terminal tape stretching from the first to the last non-blank symbol, then define

$$T(X) \equiv Y.$$

If the final tape is all blank, then define

$$T(X) = \phi.$$

If T never halts, then consider T(X) to be undefined. We will use the same symbol for this function and the Turing machine in order to emphasize the close correspondence between function evaluation and machine computation.

Let $z = \{0, 1, \ldots\}$ be the non-negative integers, and let F be a partial function from z to z. We will first define what it means for F to be evaluated by a Turing machine.

For each $n \in z$ let

$$b(n) \equiv b_k\, b_{k-1} \cdots b_1\, b_o$$

be the binary representation of the integer n; that is,

$$n = \sum_{i=o}^{k} b_i\, 2^i$$

with $b_i = 0$ or 1 and $b_k = 1$.

If there exists a Turing machine T with alphabet $A = \{B,0,1\}$ so that

$$T(b(n)) = \begin{cases} b(F(n)), & \text{if } F(n) \text{ is defined} \\ \text{undefined}, & \text{if } F(n) \text{ is undefined} \end{cases}$$

then we say that F is a Turing computable (partial) function which is evaluated by the machine T.

We will illustrate the concept of a function which can be computed by a Turing machine with the function $F(n) = n + 1$, where n is any nonnegative integer. Let n, the argument of the function, be expressed in binary form $b_k \cdots b_1\, b_o$. The input tape will appear as in Figure 2.7 with the Turing machine initially scanning the symbol b_o.

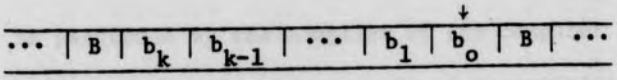


Figure 2.7 Initial Tape in Binary Form

The addition of 1 to a binary number is effected by changing digits and moving to the left until a 0 (or blank) is encountered. The final tape will contain the binary representation of F(n).

Figure 2.8 is the state diagram of such a Turing machine, M_2.

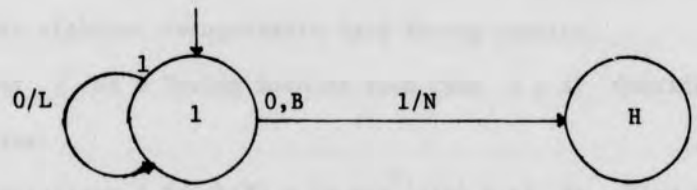


Figure 2.8 State Diagram of M_2

Modern computers can be viewed as streamlined versions of Turing machines. The modifications do not add any computational power in the sense that computers and Turing machines are able to compute exactly the same class of functions. A computer is presented with a program which describes in some formal language an algorithm which the programmer wishes the computer to follow. Programs consist of certain strings of symbols. An arbitrary string of symbols does not necessarily describe an algorithm. Those strings which do represent algorithms are called syntactically correct programs. The first operation of a computer is to check whether or not the program string is syntactically correct. This operation is performed by the compiler of the computer. The compiler is itself a modified Turing machine capable of recognizing all syntactically correct programs and of rejecting those which are not correct.

Our minds operate similar to a compiler when we converse in a natural language. Certain strings of words are grammatically correct, and others are not. Many researchers in the field of artificial intelligence believe that the human mind will eventually be modeled as a device similar to the Turing machine.

This discussion leads us to consider the notion of a language over a finite alphabet recognizable by a Turing machine.

Let T be a Turing machine such that $1 \in A$. Consider the following.

Definition 2.9: $L(T) = \{x \in A^* | T(x) = 1\}$ is called the language accepted by T.

Notice that a string x fails to be in the language accepted by T if either T never halts on x or $T(x) \neq 1$. Many writers define the language accepted by T as the set of all strings on which T eventually halts and prints 1 as the last printed symbol. We shall use this modification of the definition when we do not want to erase the string under consideration.

Definition 2.10: $L \subseteq A^*$ is called recursively enumerable (RE) if L is the language accepted by some machine T ($L = L(T)$).

Theorem 2.11: L is recursively enumerable if and only if there exists an enumeration of $L = \{y_0, y_1, \ldots\}$ and a machine T over $A' = A \cup \{0, 1\}$ so that $T(b(n)) = y_n$ for each $n \in z$.

Proof: (Sufficiency) Let $\{x_0, x_1, \ldots\}$ be a mechanical enumeration of A^* by machine M.

We shall describe the behavior of an acceptor machine P.

M generates tape x_o. P checks x_o with the list of tapes which machine T outputs. If x_o is not the same tape as any y_i, $i = 0, 1, \ldots,$ then P does not halt. If x_o is the same tape as some y_i, then P halts and prints 1. M successively generates tapes $x_1, x_2, \ldots,$ and machine P performs the same operations on each x_i as P performed on x_o. Therefore, P accepts exactly those tapes which belong to L, and so L is recursively enumerable.

(Necessity) Assume $L = L(P)$ for some acceptor machine P.

We shall describe the behavior of a machine T which enumerates L.

Let $n \in z$. For each integer k, starting with $k = 1$, T does the following:

Use M to generate x_o. Then simulate the action of P on x_o for at most k steps. If P halts and accepts x_o, then record the integer 0.

Now let M successively generate x_i, $i = 1, 2, \ldots, k$; and on each x_i let P operate for at most k steps. If P halts and accepts x_i, a check is made to determine if the integer i has been previously recorded. If not, then i is recorded.

Let T output the tape associated with the n^{th} integer recorded. This is done for each n and will produce a mechanical enumeration of L. Q.E.D.

Choose $x \in A^*$ which just happens to also be in L(T). Then T would inform us in a finite amount of time that we had in fact chosen $x \in L(T)$. But if (unknown to us) we choose $x \notin L(T)$, then the procedure defined by T may never indicate this fact, because T may never

halt on the input tape x. We certainly do not want programming languages to be susceptible to this problem. We not only want the compiler to be able to accept valid programs but also to reject invalid ones in a finite amount of time. Such languages form an important subclass called the recursive languages.

Definition 2.12: $L \subseteq A^*$ is called recursive if $L = L(T)$ for some T which eventually halts on all inputs.

Definition 2.13: $\bar{L} = \{x | x \in A^*, x \notin L\}$. That is, $\bar{L}$ is the set complement of L.

Theorem 2.14: L is recursive if and only if L and $\bar{L}$ are recursively enumerable.

Proof: (Necessity) Assume L is recursive.

Then by definition there is an acceptor machine T which halts on every $x \in L$ and prints 1. Therefore, L is recursively enumerable.

T also halts and prints 0 for every $x \notin L$. Let T' be T with output symbols 0 and 1 interchanged. T' accepts $\bar{L}$, and therefore $\bar{L}$ is recursively enumerable.

(Sufficiency) Assume L and $\bar{L}$ are recursively enumerable.

Let T_1 be an acceptor machine for L, and let T_2 be an acceptor machine for $\bar{L}$.

Construct an acceptor machine T which performs the following operations:

Select any $x \in A^*$.

First have T copy x onto another portion of the tape where it will not be erased. If the later actions of T attempt to infringe on this area, then move this copy of x over one square. Now simulate

the action of T_1 on x for one time unit. Then retrieve a copy of x, and simulate the action of T_2 on x for one time unit. Next simulate each machine for two time units, three time units, etc. Eventually one of the machines will halt and either accept or reject x. Let T act on this information so that $T(x) = 1$ if and only if $x \in L$.

Since in a finite amount of time T halts on every $x \in A^*$ and either accepts or rejects x, L is recursive. Q.E.D.

Consider associating each Turing machine $T = (A,S,f)$ with an integer as now described. Rewrite each symbol in A as an integer $0, 1, \ldots, n$ with $B = 0$; rewrite each symbol in S as $1, 2, \ldots, m$; use 0 for H, 0 for N, 1 for L, 2 for R. Let $p_o = 2$, $p_1 = 3$, $p_2 = 5$, ... be a list of the primes in increasing order.

Convert each function value

$$f(a,\delta) = (b, \delta', M)$$

to a 5-tuple of integers

$$a\ \delta\ b\ \delta'\ r \leftrightarrow n_1\ n_2\ n_3\ n_4\ n_5$$

under the above correspondence. Code each 5-tuple $n_1\ n_2\ n_3\ n_4\ n_5$ as an integer

$$N_i = p_1^{n_1} p_2^{n_2} \cdots p_5^{n_5}.$$

Arrange these integers in increasing order

$$N_1, N_2, \ldots, N_k \text{ where } k = m(n).$$

Now code the whole machine as the integer

$$g(T) = p_0^{m}\, p_1^{c(N_1)} \cdots p_k^{c(N_k)} \quad \text{where} \quad k = m(n).$$

This coding associates with each machine T a unique integer g(T) called a Gödel number for the machine.

As an example of this procedure for determining the Gödel number of a machine, consider the Turing machine M_3, which moves to the right, locates the nearest square containing the blank symbol, and halts. Figure 2.15 is the state diagram for M_3.

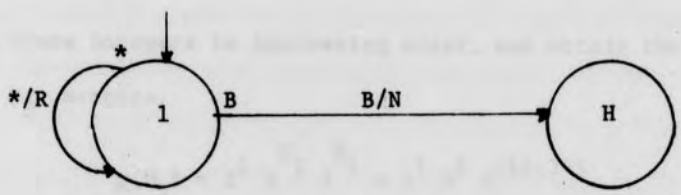


Figure 2.15 State Diagram of M_3

The computing function of M_3 is given in Table 2.16.

Input	Present State	Output	Next State	Read/write Head Movement
B	1	B	H	N
*	1	*	1	R

Table 2.16 Computing Function Table of M_3

Rewrite each symbol in the table according to the code described above.

Input	Present State	Output	Next State	Read/write Head Movement
0	1	0	0	0
1	1	1	1	2

Table 2.17 Coded Computing Function Table of M_3

Next code each line of the table as follows.

$$N_1 = 3^0\ 5^1\ 7^0\ 9^0\ 11^0$$

$$N_2 = 3^1\ 5^1\ 7^1\ 9^1\ 11^2$$

Arrange these integers in increasing order, and obtain the Gödel number of the M_3 machine.

$$g(M_3) = 2^1\ 3^{N_1}\ 5^{N_2} = 2^1\ 3^5\ 5^{114,345}$$

Note: It is a mechanical (but lengthy) process to recover the machine from its Gödel number.

Theorem 2.18: For a given alphabet A, the set $\{b(g(T)) \mid T,$ Turing machine over A$\}$ is recursively enumerable over $\{0,1\}$.

Proof: Assume that the Turing machines are numbered so that

$$g(T_1) < g(T_2) < \ldots$$

Consider the Turing machine P which performs the following task:

Pick $n \in z$ and start P on a tape containing $b(n)$.

P will start generating the integers 1, 2, ..., checking each one to see if it is the Gödel number for a valid Turing machine.

P will then output the n^{th} such integer, namely $b(g(T_n))$.

Therefore, P enumerates the set above, and by Theorem 2.11 the set is RE. Q.E.D.

Note: This procedure could be modified to output the n^{th} acceptor machine, given n.

Corollary 2.19: The set $\{b(g(T)) | T$ is an acceptor machine$\}$ is recursively enumerable.

Before stating the next theorem, we will briefly describe the concept of a universal Turing machine.

Recall that Turing's original reason for the introduction of his Turing machine concept was to provide a precise definition of the intuitive notion of an algorithm. He went further and located in the class of Turing machines a special machine, referred to as a universal machine, capable of simulating the action of any other Turing machine on any input tape. Turing then defined a procedure as mechanical or algorithmic if it could be simulated by the universal machine. We now proceed to describe the operation of the universal machine.

The complete description of a Turing machine T can be encoded on a tape. This encoding of T, say c(T), will be composed of the 5-tuples from the computing function table as pictured below.

c(T) =

a_1	δ_1	b_1	δ_1'	M_1	x	a_2	δ_2	b_2	δ_2'	M_2	x	...

Figure 2.20 Encoded Description of a Turing Machine

Let t be any input tape for T. The initial tape for the universal machine U is composed of t and c(T) as pictured in Figure 2.21.

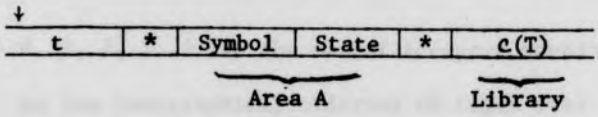


Figure 2.21 Initial Tape for the Universal Turing Machine, U

The universal machine operates in the following manner on the above tape. Initially the read/write head of U is placed on t as for the machine T. The initial state of T is stored in area A. U reads the input symbol in t, marks its place, moves to A, and prints this input symbol in the symbol section of A. U then moves to the library and searches for the 5-tuple having the same input symbol/present state pair as appears in area A. The next state for T is transferred to the state section of area A. The output symbol is printed in the marked place in t. The read/write head of U is then shifted left or right from the previously marked position as the read/write head would have moved for machine T. This procedure is then repeated until (and if) T would have halted.

Theorem 2.22: There exists a recursively enumerable subset of A^* which is not recursive.

Proof: Fix A with $0, 1 \in A$. Let P enumerate the acceptor machines over A.

That is,

$$P(b(n)) = b(g(A_n))$$

where $\{A_n | n = 0, 1, 2, \ldots\}$ is the set of acceptor machines. Let $\{x_1, x_2, \ldots\}$ be the lexigraphical ordering of tapes over A^*. Set $L \equiv \{x_n | A_n \text{ accepts } x_n\}$.

We will show that L is recursively enumerable by building an acceptor T for L. Choose $x \in A^*$. We determine n for which $x = x_n$ by listing the tapes until x appears. Machine P enumerates machines until A_n is enumerated. The rules of A_n are used on x_n in the universal machine. Then T ultimately behaves like A_n, i.e., T will accept $x_n = x$ if and only if A_n accepts x_n if and only if $x_n \in L$. Therefore, L is recursively enumerable.

We now claim that $\bar{L} = \{x_n | A_n \text{ rejects } x_n\}$ is <u>not</u> recursively enumerable. We select an arbitrary acceptor machine A_n. A_n either accepts x_n or rejects x_n. If we assume A_n accepts x_n, then $x_n \in L$. This would imply that $\bar{L} \neq L(A_n)$, the language accepted by A_n. If we assume A_n rejects x_n, then $x_n \in \bar{L}$ and $x_n \notin L(A_n)$, so $\bar{L} \neq L(A_n)$ again. If L is recursive, then $\bar{L}$ must be a language accepted by some machine. But it has been shown above that $\bar{L}$ is not accepted by an arbitrary A_n. Therefore, $\bar{L}$ is not RE, and so L is not recursive. Q.E.D.

We have indicated that the main reason Turing introduced his machine model was to give precise meaning to the notion of an "algorithm". These considerations were motivated in part by Gödel's proof of the

incompleteness of the natural numbers. One consequence of this result is the fact that, if S is the set of all logically true statements about natural numbers, then there exists no algorithm which will indicate whether or not a given statement is in S. The set S is an example of an undecidable set. We say that the decision problem for S is unsolvable. Now think of all well-formed statements about the natural numbers as written on tapes in some alphabet. Let F be the characteristic function of S; that is, $F(x) = 1$ if $x \in S$ and $F(x) = 0$ if $x \notin S$. The results above imply that F is not computable. Thus the vague question of whether we can decide algorithmically if an arbitrary statement concerning integers is or is not in S is equivalent to the more precise question of whether or not a certain function is computable. We now proceed to examine the decidability of a problem related to Turing machines called the halting problem. This problem asks the question: Given an arbitrary Turing machine T and an arbitrary input tape t, is there an algorithm which will determine whether or not T would eventually halt if started on t? This question can be converted to one concerning the existence of a certain Turing machine as follows. Let $c(T)$ and $c(t)$ be some tape encoding of the machine T and the tape t respectively. This could be the binary form of the Godel numbering described previously. Consider the following partial function.

$$D(c(T),c(t)) = \begin{cases} 1, & \text{if } T \text{ halts on } t \\ 0, & \text{if } T \text{ does not halt on } t \\ \text{undefined otherwise} \end{cases}$$

The halting problem may be rephrased as: Is D computable? Assume that it is.

Create a new machine D_1 from D such that

$$D_1(c(T),c(t)) \equiv \begin{cases} 1. & \text{does not halt} \leftrightarrow T \text{ halts on } t. \\ 2. & \text{halts and prints } 0 \leftrightarrow T \text{ does not halt on } t. \end{cases}$$

Create D_2 such that D_2 is given only the program T.

$$D_2(c(T)) \equiv \begin{cases} 3. & \text{does not halt} \leftrightarrow T \text{ halts on } c(T). \\ 4. & \text{halts and prints } 0 \leftrightarrow T \text{ does not halt on } c(T). \end{cases}$$

Now consider $D_2\ (c(D_2))$ and the following two cases.

(a) By 4, if $D_2(c(D_2))$ halts and prints 0, then D_2 does not halt on $c(D_2)$. A contradiction.

(b) If $D_2(c(D_2))$ does not halt, by 3 D_2 halts on $c(D_2)$. A contradiction.

Therefore, we conclude that decision program D does not exist, that function D is not computable, and that the halting problem is not solvable.

The halting problem is the best known member of the class of unsolvable decision problems. The unsolvability of many other members of this class is demonstrated by reducing them to the halting problem. It is shown that, if these problems are solvable, then the halting problem also would be solvable. Some examples of these unsolvable

problems are the following:

(a) Does an arbitrary machine halt on the blank tape?

(b) Does an arbitrary machine halt on every tape?

(c) Will an arbitrary machine ever print a particular symbol on an arbitrary tape?

(d) Can it be determined if two arbitrary machines compute the same function?

Researchers in machine theory have found that the definition of a Turing machine may be restricted in several ways without changing the class of computable functions. In Chapter III we will limit the alphabet to two symbols, so that the tape will contain only blank squares and/or squares with one non-blank symbol. This does not restrict the computational power of Turing machines. Multiple symbols could be coded into binary form in the two symbol alphabet. We could restrict the operations of a Turing machine by requiring that a non-blank square never be erased. Another restriction is to limit the number of states to two while retaining computational power by increasing the number of symbols in the alphabet. A tape with the doubly-infinite row of squares, which we have used throughout this chapter, may be restricted to a semi-infinite tape with information contained in alternate squares. In all of these variations the computational power of the class of Turing machines is preserved.

CHAPTER III

WANG PROGRAMS

Even though Turing machines provide a precise definition for effective procedures, the steps between the verbal statement of an algorithm and the design of a Turing machine to implement the algorithm are often difficult, obscure, and artificial. We would like to introduce an equivalent formulation of effective procedure which more closely reflects the verbal description of an algorithm. We shall present a set of symbols to be used in writing out each instruction of a procedure. Such a set of instructions corresponds closely to our sequence of thoughts as we mentally perform the same computation.

In this chapter we shall consider the relationship between a Turing machine and a programming language named for Hao Wang. A program in this language consists of five types of basic instructions and an alphabet with two symbols, a mark and a blank. As was mentioned in Chapter II, a restriction to two symbols does not lessen the computational power of the class of Turing machines. A specific sequence of instructions will replace the control unit of a Turing machine. The read/write head and tape configuration is still the same.

The five basic instructions are:

(a) R: shift the read/write head one square to the right;

(b) L: shift the read/write head one square to the left;

(c) *: mark the square of the tape under scan;

(d) E: erase the square under scan;

(e) C(x): conditional transfer such that if the square under scan is marked, then follow the instruction labeled x; otherwise, follow the next instruction in the sequence.

An added instruction, H, indicates an end to computation.

A program is a sequence of these instructions with successive instructions labeled with consecutive integers. The reader will notice that, when an instruction is *, E, or C(x), the read/write head does not move from the square currently scanned. Only when an instruction of R or L is reached in the program, does the read/write head shift position.

We now present two examples of a Wang program and the graphical form of a corresponding Turing machine. In the first example the read/write head is instructed to mark its present position and then move right until it locates and halts on the first square containing the blank symbol. Table 3.1 contains the Wang program, W_1, and the Turing machine is diagrammed in Figure 3.2.

1. *
2. R
3. C(2)
4. H

Table 3.1 Wang Program, W_1

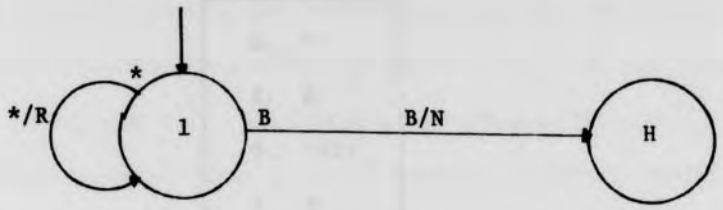


Figure 3.2 Turing Machine for W_1

The second example, W_2, instructs the Turing machine to compute the function $f(x_1,x_2) = x_1 + x_2$, where x_1 and x_2 are any nonnegative integers and are represented on the tape by $x + 1$ consecutive *'s. The input tape will appear as in Figure 3.3 with the Turing machine initially scanning the leftmost symbol of x_1.

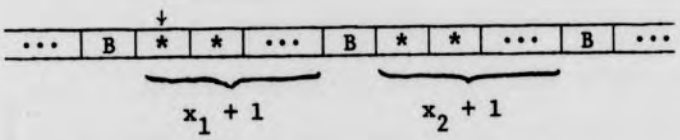


Figure 3.3 Initial Tape for W_2

The output tape will contain $f(x_1, x_2)$, represented by $x_1 + x_2 + 1$ consecutive *'s. Table 3.4 contains the Wang program, W_2, and the Turing machine is diagrammed in Figure 3.5. The reader should convince himself that the program in Table 3.4 will perform the required task.

One may wish to have the ability to make a conditional transfer when the square under scan is blank. To avoid the addition of another instruction to our set of basic instructions, we introduce the concept of a subroutine. This allows us to make a transfer on a blank square

1.	*
2.	R
3.	C(1)
4.	*
5.	R
6.	C(4)
7.	L
8.	E
9.	L
10.	E
11.	H

Table 3.4 Wang Program, W_2

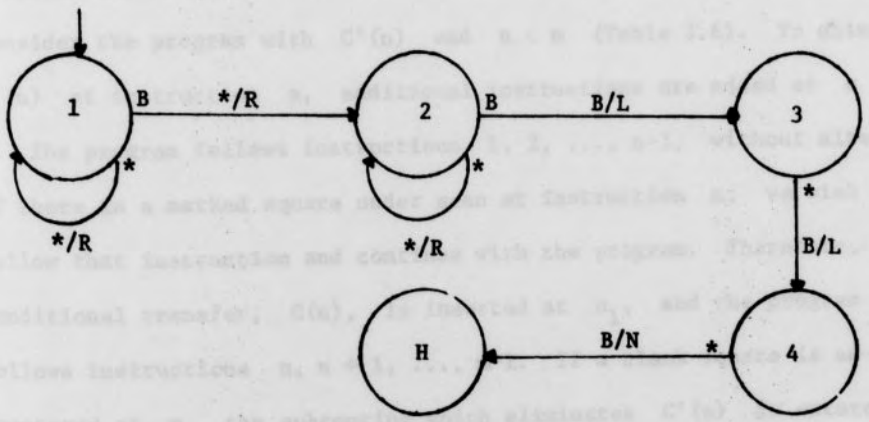


Figure 3.5 Turing Machine for W_2

while preserving the original set of instructions. We denote the conditional transfer on a blank square by C'(x). If the square under scan is blank, the x^{th} instruction is followed; otherwise, the next instruction on the list is followed. Assume that in a particular program one wishes to have the conditional transfer on a blank square made at instruction m; i.e., m. C'(n). If $n < m$, then a transfer is made to a prior instruction in the program. If $n > m$, then a transfer is made to an instruction which follows in the program. In the case of $n = m$, the program will enter an endless loop at statement m if a blank is encountered. Although the machine never enters the halting state, computation effectively ceases.

Tables 3.6, 3.7, and 3.8 contain discriptions of a program with C'(n) and the same program with C'(n) eliminated by use of a subroutine. A subroutine refers to a program block designed to perform a specific task. Such a block can be inserted into larger programs. Consider the program with C'(n) and $n < m$ (Table 3.6). To eliminate C'(n) at instruction m, additional instructions are added at n and m. The program follows instructions 1, 2, ..., n-1, without alteration. If there is a marked square under scan at instruction n, we wish to follow that instruction and continue with the program. Therefore, a conditional transfer, C(n), is inserted at n_1, and the program follows instructions n, n + 1, ..., m-1. If a blank square is encountered at m, the subroutine which eliminates C'(n) is entered. The square is marked (m_2), a conditional transfer is made to n_2 where the mark is erased, and the program continues. If a marked

square is encountered at m, instruction m_1 contains a transfer to m + 1, and the program then continues. The programs in Tables 3.7 and 3.8 contain similar alterations in the elimination of C'(n).

The Wang programming language is a simplified version of the languages used in computer programs, which have greatly enlarged instruction sets. Although the Wang language consists of only five basic instructions, the erasing instruction could be eliminated from the set without limiting the computational power of this class of Turing machines.

In Chapter II it was noted that Turing machines can be viewed as simplified versions of modern computers. With the description of the Wang programming language added to the definition of a Turing machine, we have the basic elements of the more sophisticated computer operations.

Program with C'(n)	Program without C'(n)
1.	1.
.	.
.	.
.	.
n - 1.	n - 1.
	n_1. C(n)
n.	n_2. E
	n.
n + 1.	n + 1.
.	.
.	.
.	.
m - 1.	m - 1.
	m_1. C(m+1)
m. C'(n)	m_2. *
	m_3. $C(n_2)$
m + 1.	m + 1.
.	.
.	.
.	.

Table 3.6 Elimination of C'(n) from a Wang Program with $n < m$

Program with C'(n)	Program without C'(n)
1. . . . m - 1.	1. . . . m - 1.
m. C'(n) {	m_1. C(m + 1) m_2. * m_3. C(n_2)
m + 1. . . . n - 1.	m + 1. . . . n - 1.
n. {	n_1·C(n) n_2. E n.
n + 1. . . .	n + 1. . . .

Table 3.7 Elimination of C'(n) from a Wang Program with n > m

Program with C'(n)	Program without C'(n)
1. . . . m - 1. m. C'(n) m + 1. . . .	1. . . . m - 1. m_1. C(m + 1) m_2. E m_3. * m_4. C(m_2) m + 1. . . .

Table 3.8 Elimination of C'(n) from a Wang Program with n = m

CHAPTER IV

GRAMMARS

In Chapter II we illustrated how Turing machines are used as acceptors for recursively enumerable languages. But consider how you proceed to understand the semantic content of an English sentence which is grammatically (syntactically) correct. As part of the syntactic analysis, the sentence is divided into interrelated phrases (subject, verb, modifiers, etc.), each with its own meaning. Turing machines provide poor models for the analysis of sentences in the natural languages and the programming languages, because they do not clearly indicate the phrase-structure of the sentence. Grammars, however, parse (divide) a string of symbols in much the same way as high school students diagram a sentence in the English language. The following schematic illustrates parsing of an English sentence in general form: (Sentence) → (Subject)(Verb)(Object) → etc.

To define a grammar we first specify an alphabet A, consisting of two disjoint subsets, the terminal alphabet T and the non-terminal alphabet N. The terminal alphabet consists of the set of symbols which can occur in the sentences of the language. In terms of diagramming a sentence in the English language, these symbols correspond to the specific words which constitute the sentence. The non-terminal alphabet consists of symbols which ultimately represent groups of terminal symbols. The non-terminal alphabet for English could consist of such symbols as

S for (sentence), V for (verb), O for (object), etc. The substitution of one group of symbols for another group of symbols is defined by a set of production rules P. In diagramming an English sentence, the non-terminal alphabet corresponds to such terms as noun phrase, verb phrase, article, adjective, noun, verb, adverb, etc. For any grammar a particular non-terminal symbol S serves as the initial symbol, or sentence symbol, in the substitution process. These four elements - terminal and non-terminal alphabets, productions, and the distinguished initial symbol - define a grammar.

In formal terms a grammar $G = (T,N,S,P)$ is a system where

(a) T is a finite set called the terminal alphabet of G.

(b) N is a finite set called the non-terminal alphabet of G

$A = T \cup N$ $(T,N$ disjoint) is called the alphabet of G.

(c) $S \in N$ is the initial symbol of G.

(d) $P \subset A^{+} \times A^{*}$ is a finite set of ordered pairs with each pair of P called a production of G.

The notation $x \rightarrow y$ will denote the fact that $(x,y) \in P$ and that the string y may be substituted for string x, where x may be a substring of a larger string. If $|x| > |y|$, then the production $x \rightarrow y$ is called an erasing rule.

As an example of a grammar, let $G = (T, N, S, P)$ where $T = \{a,b\}$, $N = \{S,B\}$, and P consists of the following productions:

(1) $S \rightarrow \phi$ (ϕ denotes the empty string.)

(2) $S \rightarrow SB$

(3) $B \rightarrow b$

By applying these productions to the initial symbol S, we obtain the following (The number of the production which was applied at each step is indicated above the $\rightarrow$ symbol.):

(a) $S \overset{(1)}{\rightarrow} \phi$

Since $|S| > |\phi|$, production 1 is an erasing rule.

(b) $S \overset{(2)}{\rightarrow} SB \overset{(1)}{\rightarrow} B \overset{(3)}{\rightarrow} b$

(c) $S \overset{(2)}{\rightarrow} SB \overset{(2)}{\rightarrow} SBB \overset{(1)}{\rightarrow} BB \overset{(3)}{\rightarrow} bB \overset{(3)}{\rightarrow} bb$

By iterating the application of production 2, any desired number of b's in a string may be obtained.

A sequence of strings, $X_1, X_2, X_3, \ldots, X_n$, over A in which each string X_{i+1} in the sequence is produced from the previous string X_i by using one production rule is called a <u>derivation</u> and denoted $X_1 \underset{G}{\Longrightarrow} X_n$. The production rules may be applied to substrings of the strings X_i.

Each sequence of strings in (a), (b), or (c) is called a derivation. In (c), for example, the derivation is denoted $S \underset{G}{\Longrightarrow} bb$.

Any string $y \in T^*$ such that $S \underset{G}{\Longrightarrow} y$ is called a <u>sentence</u>. The set $L(G) = \{y \in T^* \mid S \underset{G}{\Longrightarrow} y\}$ is called the <u>language</u> of the grammar. In the above example, the language of the grammar is $L(G) = \{b\}^* \subseteq \{a,b\}^*$.

Since the strings in L(G) can be effectively enumerated by listing them in increasing length of derivation, L(G) is a recursively enumerable set. In fact, one can show that the set {L(G) | G, grammar with terminal alphabet T} is precisely the set of recursively enumerable languages over T.

Each production $x \rightarrow y$ may be thought of as the substitution of a phrase or group of phrases for an existing phrase. In this way the derivation of a sentence serves as a parsing of the sentence into interrelated phrases. This fact has been used to advantage in designing grammars for programming languages. The parsing process is an important first step toward understanding the meaning of a sentence.

As we mentioned in Chapter II, the nature of programming languages requires that we be able to decide in a finite amount of time whether or not a particular program is a valid one. Therefore, we need grammars which we can guarantee will generate languages which are recursive sets.

In 1956 Noam Chomsky introduced a set of restrictions on the productions of a grammar, and the resulting classification of grammars is known as the Chomsky Hierarchy. The unrestricted grammars, or type 0 in the hierarchy, are the grammars which we have previously described with no restrictions on P and which generate languages which are recursively enumerable sets.

The remaining classes of grammars in the hierarchy are subsets of the set of unrestricted grammars, and the languages are recursive sets. Each class is a subset of all previously listed classes. There is a single restriction on the productions which applies to all of the grammars in these classes. We first require that there be no erasing rule, except possibly $S \rightarrow \phi$. Therefore, all productions $x \rightarrow y$ are such that $|x| \leq |y|$.

The grammars of type 1 are called the <u>context-sensitive grammars</u>. In addition to the erasing rule restriction, productions are required to

be of the form $wBz \rightarrow wxz$, where $w,x,z \in A^*$, $x \neq \phi$, and B is a single non-terminal symbol. Thus, x can only be substituted for B in the context of w on the left and z on the right.

The productions of the context-sensitive grammars, however, are not restricted enough to be of practical use for programming languages, but some programming languages can be characterized as type 2 or context-free grammars. A type 2 grammar has the erasing rule restriction, and in addition each production is of the form $B \rightarrow x$, where B is a single non-terminal symbol and $x \in A^+$. Note that with $w = z = \phi$ in a type 1 grammar, we have a type 2 production rule. Therefore, each context-free grammar is a context-sensitive grammar.

Context-free languages are sufficiently restricted and yet remain powerful enough to meet the requirements of programming languages. ALGOL 60 was the first programming language, defined by a context-free grammar. Earlier programming languages were developed without application of language structuring devices, and the analysis of a program depended upon a special set of rules. Today compilers use syntatic analysis techniques which are based upon formal grammars.

The grammars of type 3, or the regular grammars, are too restricted to have any practical applications for programming languages. Regular grammars have the erasing rule restriction, and the productions of this class are of the form $B \rightarrow bC$ or $B \rightarrow b$, where B,C are single non-terminal symbols and b is a single terminal symbol.

The languages generated by each type of grammar are exactly those languages which are accepted by a certain class of automata. The

unrestricted grammars generate languages which are accepted by Turing machines. The regular grammars generate languages which are accepted by finite-state machines. Turing machine-like structures have been found as acceptors for type 1 and type 2 grammars. These acceptors provide an efficient means for determining whether or not a string of symbols belongs to the language generated by a particular grammar. However, they fail as analyzers of the phrase structure of the sentences. Grammars provide a diagram of the successive steps in the analysis of a sentence. However, they perform poorly in determining whether the sentence belongs to a particular language, for they require the use of a trial and error procedure. This method is known as bottom-up parsing. The sentence is searched for substrings which are the right parts of the given productions. These substrings are replaced by the corresponding left parts of the productions. The productions are thus applied backwards until, if possible, the initial symbol is reached. Error occurs if one chooses the wrong production rule from several possible choices.

CHAPTER V

GRIDS

In our discussion of machines and grammars we have been concerned only with a single tape or string of symbols. We will now consider the relationship between machines, grammars, and a natural two-dimensional generalization of a tape, which we will call a grid. A grid G is an infinite collection of tapes,arranged successively as rows in an infinite array. Although we shall consider only two-dimensional arrays, the ideas in this chapter could be extended to n-dimensional arrays.

The machine which operates on a grid is defined in the same way as a Turing machine. The grid machine consists of three parts, as represented schematically in Figure 5.1:

All but a finite number of the squares of a grid are initially blank. The marked squares may contain any symbol from a given finite alphabet.

The grid machine operates sequentially on the grid as does a Turing machine with the following exception. After the initial state of the control unit is changed, the read/write head moves one square in either of four directions - up, down, to the left, or to the right.

A formal definition of a grid machine follows. The triple $G = (A, S, f)$ is a grid machine if:

(a) A is a finite nonempty set of alphabet symbols which contains a distinguished symbol B called a blank.

(a) a control unit

(b) a read/write head

(c) an unbounded array of squares used for input and output.

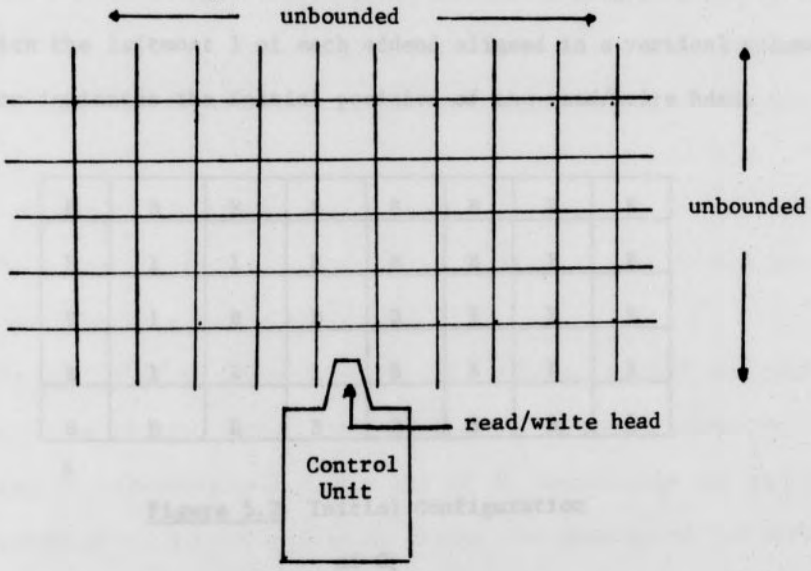


Figure 5.1 Grid Machine

(b) S is a finite nonempty set of symbols called states. The distinguished symbol H, which is not in S, is called the halting state.

(c) f is a function such that

$f:A \times S \to A \times (S \cup \{H\}) \times \{U, D, R, L, N\}$

with the property that $\delta' = H$ if and only if $M = N$ in $f(a,\delta) = (b,\delta',M)$.

Consider a grid machine G_1 which prints the sum of an arbitrary finite number of positive integers, represented on the grid in unary form. As shown in Figure 5.2, the addends are arranged in successive rows with the leftmost 1 of each addend aligned in a vertical column. An arrow indicates the initial position of the read/write head.

B	B	B	B	B	B	B	B
B	1	1	B	B	B	B	B
B	1	B	B	B	B	B	B
B	1	1	1	B	B	B	B
B	B	B	B	B	B	B	B

↑

Figure 5.2 Initial Configuration of G_1

In Figure 5.3 the sum in unary form appears in the row below the addends, marked by *. The arrow indicates the position of the read/write head at the end of computation.

B	B	B	B	B	B	B	B
B	1	1	B	B	B	B	B
B	1	B	B	B	B	B	B
B	1	1	1	B	B	B	B
*	1	1	1	1	1	1	B

↑

Figure 5.3 Final Configuration of G_1

The machine operates as follows. The read/write head of G_1 moves up one square from its initial position, reads each 1 appearing in the first addend, and transfers it to the row where the sum is to appear. This procedure is followed for every addend printed on the grid. The resulting row of 1's is the sum in unary form.

The graphical form of a particular machine G_1 which implements this algorithm is given in Figure 5.4.

The notion of an acceptor machine is still meaningful for grid machines. We simply require that the machine M print either an accepting or rejecting symbol when and if M halts. The set of all grids accepted by a grid machine is called the language of the machine and is denoted L(M).

For each grid $G \in L(M)$ the machine necessarily halted on G in a finite amount of time. Therefore, the machine examined only a bounded portion of the grid G. We may insert any symbols we wish into the

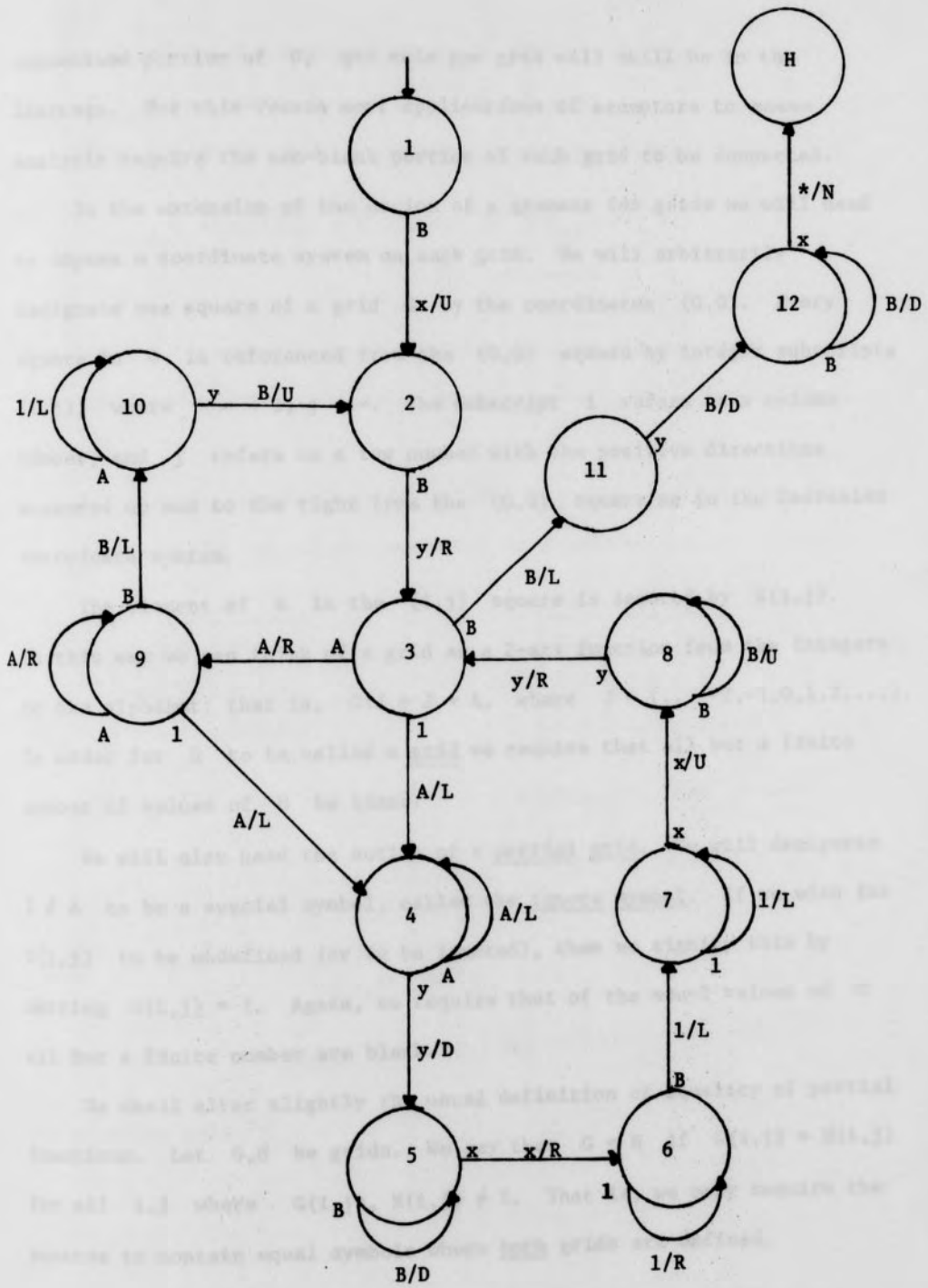


Figure 5.4 Graphical Form of G_1

unexamined portion of G, and this new grid will still be in the language. For this reason most applications of acceptors to scene analysis require the non-blank portion of each grid to be connected.

In the extension of the notion of a grammar for grids we will need to impose a coordinate system on each grid. We will arbitrarily designate one square of a grid G by the coordinates $(0,0)$. Every square in G is referenced from the $(0,0)$ square by integer subscripts (i,j), where $-\infty < i, j < \infty$. The subscript i refers to a column number, and j refers to a row number with the positive directions measured up and to the right from the $(0,0)$ square as in the Cartesian coordinate system.

The element of A in the (i,j) square is denoted by $G(i,j)$. In this way we can think of a grid as a 2-ary function from the integers to the alphabet; that is, $G:J \times J \to A$, where $J = \{\ldots -2,-1,0,1,2,\ldots\}$. In order for G to be called a grid we require that all but a finite number of values of G be blank.

We will also need the notion of a partial grid. We will designate $I \notin A$ to be a special symbol, called the ignore symbol. If we wish for $G(i,j)$ to be undefined (or to be ignored), then we signify this by setting $G(i,j) = I$. Again, we require that of the non-I values of G all but a finite number are blank.

We shall alter slightly the usual definition of equality of partial functions. Let G,H be grids. We say that $G = H$ if $G(i,j) = H(i,j)$ for all i,j where $G(i,j), H(i,j) \neq I$. That is, we only require the squares to contain equal symbols where both grids are defined.

We say that F is a <u>subgrid</u> of G if there exists k,m such that $F_{k,m} = G$, where $F_{k,m}(i,j) \equiv F(i + k, j+m)$.

Given two grids F,G we define <u>composition</u> of grids by

$$(F \cdot G)(i,j) = \begin{cases} G(i,j) & \text{if } F(i,j) = I \\ F(i,j), & \text{otherwise} \end{cases}$$

We can think of composition as actually printing the values of F into the corresponding squares of G, except where F is undefined (has value I).

For example, consider grids G and F below, and let A = {B,a,b}. We will designate the (0,0) square of each grid by • .

B	B	B	B
B	•a	a	B
B	B	B	B

<u>Figure 5.5</u> Grid G

I	I	I	I
I	•a	b	I
I	I	I	I

<u>Figure 5.6</u> Grid F

B	B	B	B
B	•a	b	B
B	B	B	B

<u>Figure 5.7</u> Grid F•G

Now consider grid $F_{0,1}$.

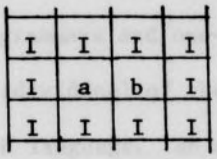


Figure 5.8 Grid $F_{0,1}$

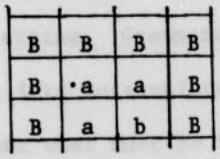


Figure 5.9 $F_{0,1} \cdot G$

One can show that composition is associative, i.e., $(F \cdot G) \cdot H = F \cdot (G \cdot H)$.

Proof:

$$[(F \cdot G) \cdot H](i,j) = \left\{ \begin{array}{ll} G(i,j) & \text{if } F(i,j) = I \\ F(i,j) & \text{otherwise} \end{array} \right\} . \; H(i,j)$$

$$= \left\{ \begin{array}{ll} H(i,j) & \text{if } F(i,j) = G(i,j) = I \\ G(i,j) & \text{if } F(i,j) = I \neq G(i,j) \\ F(i,j) & \text{if } F(i,j) \neq I \end{array} \right.$$

$$[F \cdot (G \cdot H)](i,j) = F(i,j) \cdot \left\{ \begin{array}{ll} H(i,j) & \text{if } G(i,j) = I \\ G(i,j), & \text{otherwise} \end{array} \right\}$$

$$= \left\{ \begin{array}{ll} H(i,j) & \text{if } F(i,j) = G(i,j) = I \\ G(i,j) & \text{if } F(i,j) = I \neq G(i,j) \\ F(i,j) & \text{if } F(i,j) \neq I \end{array} \right.$$

When we considered grammars and one-dimensional strings of symbols, we compared the parsing (dividing) of the string to the diagramming of a sentence in the English language. An acceptor machine efficiently determines whether or not a string of symbols belongs to the language generated by a particular grammar, but it fails in the analysis of the phrase structure of the string. Grammars provide a convenient model of the division of a string into interrelated phrases.

Consider for a moment what must be the mental processes employed in recognizing hand-written letters of the alphabet. The human mind is capable of distinguishing one letter from another even when the strokes which compose each letter are greatly distorted. Each of us must be "programmed" to identify certain essential interrelationships among the strokes and to ignore certain distortions. An extension of the notion of grammars to two-dimensional grids could provide an efficient model for displaying the interrelationships which exist among the elements of a grid.

In grid grammars we shall specify an alphabet A, which consists of two disjoint subsets, the terminal alphabet T and the non-terminal alphabet N. These alphabets will serve the same purpose as for one-dimensional grammars.

In applying a production $x \to y$ of a one-dimensional grammar to a string wxz, we first locate x as a substring of wxz and then replace x with y. If $|x| < |y|$, then this substitution requires widening the segment occupied by x so that y can be accommodated. Our first impulse then is to extend this notion to grids as follows.

In a production rule $F \rightarrow G$, let F and G be bounded segments of grids. If the segment F occurs in a grid H, then replace F with G. However, if F and G don't have the same geometric shape, G cannot replace F without distorting the relationships of the alphabet symbols of H. Therefore, we would have to restrict ourselves to isotonic grid grammars, that is, productions which have geometrically identical left and right members. On the other hand, the notions of partial grids and composition of grids discussed above provide a means for substituting one grid segment for another segment of a different shape.

In two-dimensional grammars we will define a production rule as an ordered pair of partial grids over the combined alphabet $A = T \cup N$. The application of a production rule to a grid will utilize the notion of composition of grids defined above. Given grids F,G, and H we determine whether the production $F \rightarrow G$ may be applied to H in the following way. First we determine if there exists some k,m such that $F_{k,m} = H$. If so, then $(G_{k,m} \cdot H) \equiv H'$ will be the resulting grid when the production rule $F \rightarrow G$ is applied to H.

We will designate the <u>initial grid</u> S to be the grid with all blank squares.

Formally, a grid grammar $G = (T,N,S,P)$ is defined to be a system where

(a) T is a finite set called the <u>terminal alphabet</u> of G.

(b) N is a finite set called the <u>non-terminal alphabet</u> of G. $A = T \cup N$ (T,N disjoint) is called the <u>alphabet</u> of G.

(c) S is the initial grid of G.

(d) P is a finite set of ordered pairs of partial grids with each pair $(F,G) \in P$ called a production of G and denoted $F \rightarrow G$.

As an example, consider a grammar which produces only grids which contain a single vertical line. Let $G_1 = (T,N,S,P)$ where $T = \{\blacksquare, B\}$, $N = \{*\}$, and P consists of the productions listed in Figure 5.10.

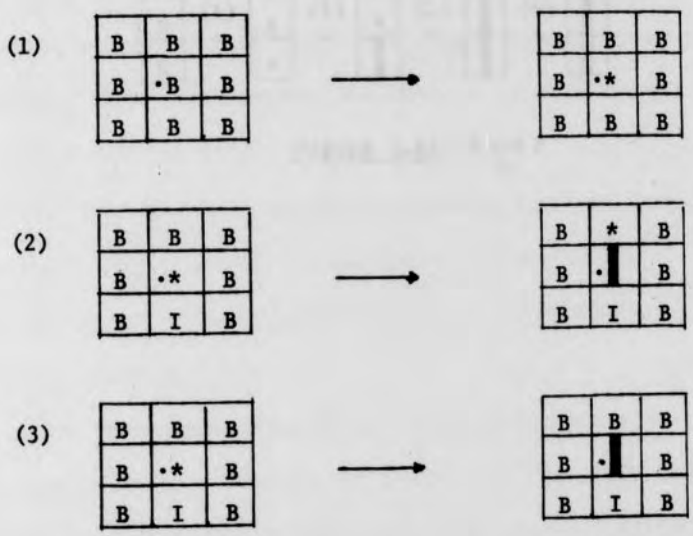


Figure 5.10 Production Rules of G_1

A sequence of grids $F_1, F_2, \ldots, F_n$ over A in which each grid F_{i+1} in the sequence is produced from the previous grid Fi by using one production rule is called a derivation and denoted $F_1 \underset{G}{\Longrightarrow} F_n$.

Any grid F such that $S \underset{G}{\Longrightarrow} F$ and such that every alphabet symbol of F is a terminal symbol will be called a scene of the grammar. The

set of all scenes produced by a particular grammar G is called the language of G and denoted L(G).

In the example above, the language of G_1 is the set of all grids containing a single vertical line. Figure 5.11 illustrates the derivation of one scene F belonging to $L(G_1)$. The number of the production which was applied at each step is indicated above the → symbol.

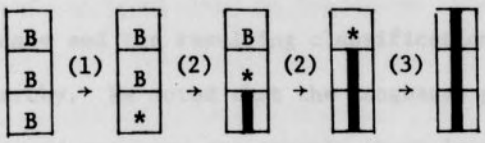


Figure 5.11 $S \underset{G_1}{\Longrightarrow} F$

CHAPTER VI

RESTRICTED GRID GRAMMARS

In this chapter we will introduce a set of restrictions on the left numbers of the productions of a grid grammar. In Chapter IV we discussed the restrictions on the productions of a grammar in the one-dimensional case and the resulting classification of grammars, the Chomsky Hierarchy. We noted that the languages generated by each of the four types of grammars are exactly those languages which are accepted by a certain class of automata. A comparable relationship between grid grammars and grid automata would be desirable, but research over the past decade has failed to establish an equivalence between a particular formulation of a two-dimensional grammar and a natural array machine.

The restrictions which we will impose affect one of the two types of regions of the left member of each of the production rules. A region is any subset of the squares of a grid. Since a grid is defined as a partial function from pairs of integers into the alphabet, the entry in a particular square may be an alphabet symbol or the symbol I if the grid function is undefined for that square. The defined region of a grid will be referred to as the non-ignore (non-I) region. The function of the I symbol in the application of production rules was discussed in Chapter V with an example illustrated in Figures 5.10 and 5.11.

An arbitrary grid has no restriction placed on the location of these regions on the grid and no limit placed on their number, their size, or their shape. Now let us consider what happens if we place restrictions on the non-I regions of each left member of the production rules.

It seems natural to restrict the non-I region of a production rule as a means of restricting the associated grammar. The non-I portion must be located and identified in the master grid before a production rule can be applied. One should keep in mind the analogy of a plane flying over a partially clouded landscape. It is easier to identify the locality below if the break in the clouds is one large hole rather than a number of smaller holes. Therefore, we will first require that the non-I regions be connected, thus reducing the number of components of these regions to one.

Since the value of every square in the non-I region must be identified, we must be able to scan every square in the non-I region in a finite amount of time. A second restriction will be to require that the non-I region consist of only a finite number of the squares of the grid.

The shape of the non-I region also affects our ability to locate it in another grid. Consider a region R with the following property. For each two squares of R which are the endpoints of a vertical or horizontal sequence of squares, all of the intermediate squares are in R. We thus alter the usual definition of convexity and define R to be <u>convex</u>. The squares of R will be easier to scan than the squares of a non-convex region. A third restriction will be to require that the non-I region be convex.

We now have a non-I region which is finite and convex. A fourth restriction will be to limit the non-I region to at most two symbols (or squares) with at least one of them a non-terminal symbol. A final restriction will be to limit the non-I region to a single non-terminal.

The reader will note that we have produced a hierarchy of increasingly severe restrictions on the non-I regions of the left members of the production rules. The classification of these left members may be described as follows:

(1) A grid whose non-I region is arbitrary.

(2) A grid whose non-I region is connected.

(3) A grid whose non-I region is connected and finite.

(4) A grid whose non-I region is connected, finite and convex.

(5) A grid whose non-I region is connected and consists of at most two symbols with at least one of them a non-terminal.

(6) A grid whose non-I region consists of a single non-terminal symbol.

If a grammar is produced from a set of production rules satisfying restriction (i) above, then we will refer to it as an i-grammar.

We will define equivalent grammars to be two grammars over the same terminal alphabet which generate the same language.

6.1 Theorem: Any 3-grammar is equivalent to a 3-grammar with the non-I region of all left members of the production rules simply connected.

Proof: In the proof of this theorem each production rule will be replaced with a finite number of production rules satisfying the restriction of the theorem in such a way that the same language is produced.

Let $G_1 = (T, N, S, P)$ be a 3-grammar and let $(F,G) \in P$. Let R be the non-I region of F. Enclose R with the smallest simply connected region possible, and label this region S. Since R is finite, it is possible to enclose R with a simply connected finite region. Let R* be the region of S, having squares with the value I. Let m be the number of squares in R*. Let n be the size of the alphabet $A = T \cup N$.

Let F_i, $1 \leq i \leq n^m$, agree with F except that each F_i represents a different assignment of alphabet symbols to the squares of R*. Let the set of production rules $\{F_1 \rightarrow G, F_2 \rightarrow G, \ldots, F_{n^m} \rightarrow G\}$ replace (F,G) in P.

Repeat the above procedure for every production in P, and let P' be the derived set of new production rules. Then $G_2 = (T, N, S, P')$ is a 3-grammar whose production rules have left members with a non-I region that is simply connected.

In G_1 if (F,G) is applied, then this application corresponds to some (F_i,G) in G_2. Conversely, if some (F_i,G) in G_2 is applied. Then this application is the same as (F,G) in G_1. Therefore, the same scenes are produced, and $L(G_1) = L(G_2)$. Q.E.D.

6.2 Theorem: Each 3-grammar is equivalent to a 4-grammar.

Proof: We will show that each production rule can be replaced by a sequence of production rules in which the concave gaps in the original left members are filled with all possible combinations of values from the alphabet in such a way that no new scenes will be produced by the new grammar.

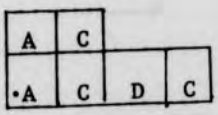
Let $G_1 = (T, N, S, P)$ be a 3-grammar, and let $(F,G) \in P$. Let R be the non-I region of F. Enclose R with its convex hull C. Since the convex hull of R is the smallest convex region containing R and R is finite, C is finite. Let R* be the ignore region of C; that is, the relative difference of R in C. Let m be the number of squares in R*. Let n be the size of the alphabet $A = T \cup N$.

Let F_i, $1 \leq i \leq n^m$, agree with F except that each F_i represents a different assignment of alphabet symbols to the squares of R*. Let the set of production rules $\{F_1 \rightarrow G, F_2 \rightarrow G, \ldots, F_{n^m} \rightarrow G\}$ replace (F,G) in P.

Repeat the above procedure for every production in P, and let P' be the derived set of new production rules. Then $G_2 = (T, N, S, P')$ is a 4-grammar whose production rules have left members with a non-I region that is convex.

In G_1 if (F,G) is applied, then this application corresponds to some (F_i,G) in G_2. Conversely, if some (F_i,G) in G_2 is applied, then this application is the same as (F,G) in G_1. Therefore, the same scenes are produced, and $L(G_1) = L(G_2)$. Q.E.D.

Consider how a production rule for a 4-grammar may be replaced by a set of production rules for a 5-grammar. Let $G_1 = (T, N. S. P)$ be a 4-grammar where $N = \{A, C, D\}$. Let $(F,G) \in P$, and let the non-I region of F be



Consider the set of productions in Figure 6.3.

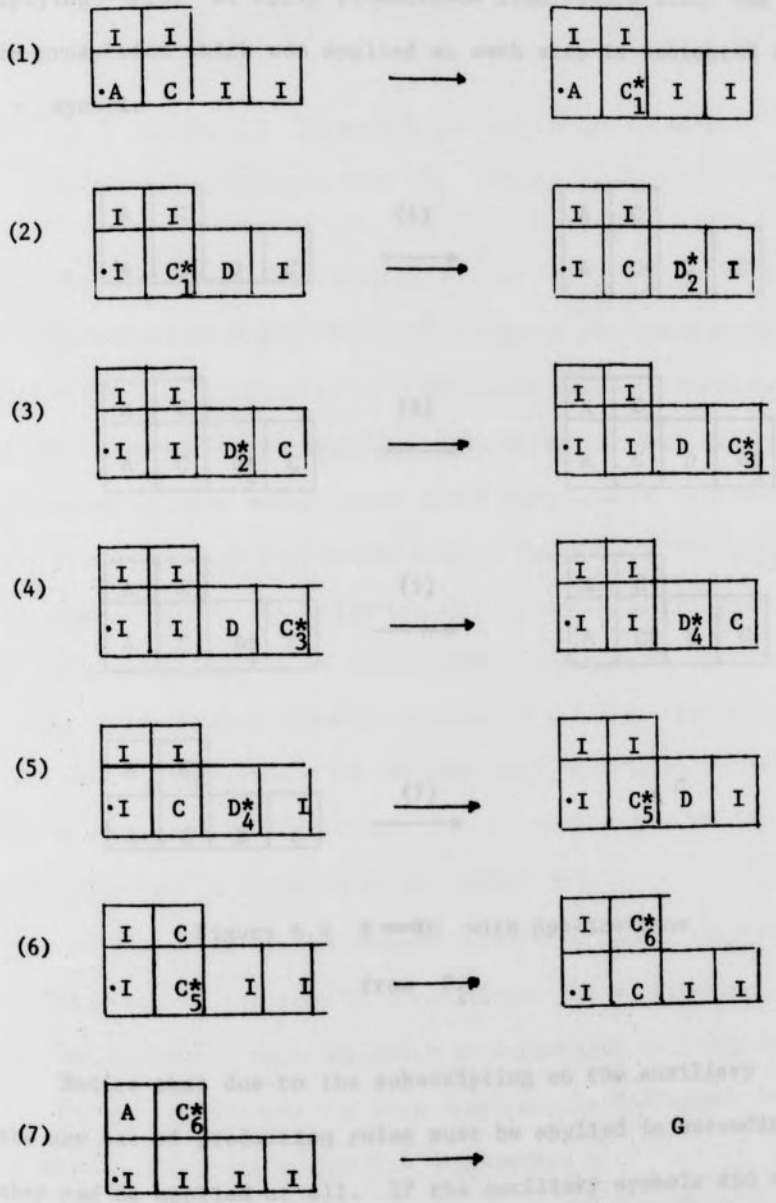


Figure 6.3 Production Rules P_1

Suppose we have located F in a master grid, and instead of applying (F,G) we apply productions from Figure 6.3. The number of the production which was applied at each step is indicated above the → symbol.

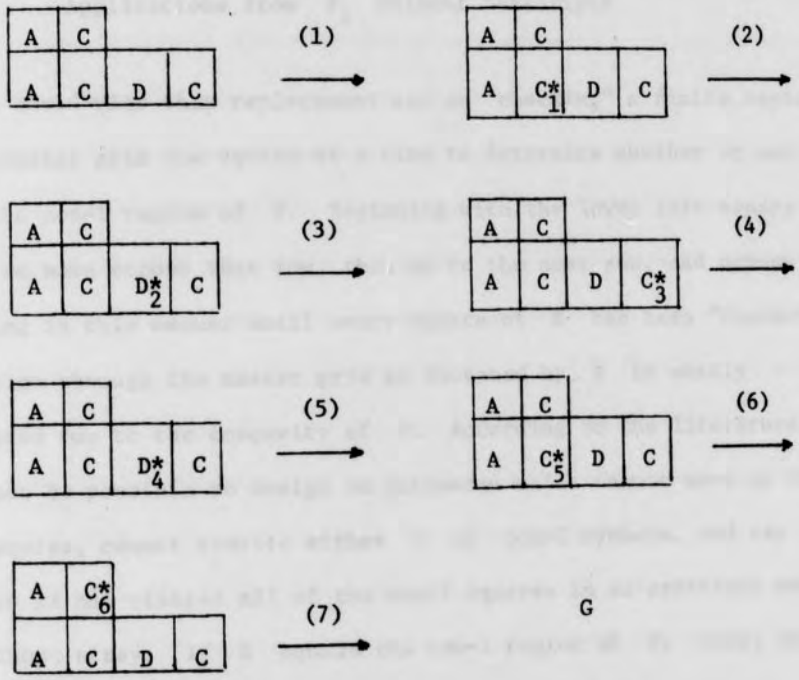


Figure 6.4 F ⟹ G with Applications from P_1

Notice that due to the subscripting on the auxiliary * symbols, the new set of production rules must be applied in ascending order, if they can be applied at all. If the auxiliary symbols did not contain subscripts, the derivation in Figure 6.5 could have occurred.

Figure 6.5 Incorrect Derivation of G with Applications from P_1 without Subscripts

One could view this replacement set as "checking" a finite region R of a master grid one square at a time to determine whether or not R equals the non-I region of F. Beginning with the lower left square of R, we move across that row, then up to the next row, and across it, continuing in this manner until every square of R has been "checked". This motion through the master grid as dictated by F is easily implemented due to the concavity of F. According to the literature, it may not be possible to design an automaton which cannot move on the ignore squres, cannot rewrite either I or non-I symbols, and can know when it has visited all of the non-I squares in an arbitrary non-convex input array. If R equals the non-I region of F, then, and only then, is G printed on the master grid.

The following theorem demonstrates that each production rule of a 4-grammar may be replaced by a restricted set as illustrated above.

6.6 Theorem: Each 3-grammar is equivalent to a 5-grammar.

Proof: By Theorem 6.2 each 3-grammar is equivalent to a 4-grammar, so we will prove the result for 4-grammars.

Let $G_1 = (T,N,S,P)$ be a 4-grammar, and let $(F,G) \in P$. Let R be the non-I region of F, and assume that the leftmost square of the lowest row of R is square (i,j).

We will describe a set of production rules which will replace (F,G) in P. The reader should consider each new production rule as a movement from one square of R to another as in the above example. We will replace (F,G) with at most $2 \cdot |R|$ production rules which will be labeled (F_k, G_k), $k = 1, 2, \ldots$. The subscript k should be increased consecutively throughout the creation of all production rules so that no new rules have the same subscript. The creation of (F_k, G_k) will require the introduction of new non-terminal symbols into the alphabet. The notation we shall adopt for these auxiliary symbols is the following. If we wish to retain any symbol A which appears in F and which is to be replaced by a new symbol in creating (F_k, G_k), then we will convert A to the auxiliary symbol A_k^*. As in the new production rules, the subscript k will be increased consecutively throughout the creation of all auxiliary symbols. The reader should note the introduction of auxiliary symbols in the above example.

(1) We will first describe a set of production rules which enable us to move right from square (i,j) of F across the squares of R which are in the j^{th} row of F. If $F(i + \ell, j) \neq I$, $\ell = 1, 2, \ldots$, create the following set of productions: $F_k \rightarrow G_k$ where F_k, G_k are grids with values of I except

if $\ell = 1$, then

$F_k(i,j) = F(i,j)$

$F_k(i + 1,j) = F(i + 1,j)$

if $\ell > 1$, then

$$F_k(i + \ell - 1, j) = G_{k-1}(i + \ell - 1, j)$$

$$F_k(i + \ell, j) = F(i + \ell, j)$$

In both cases

$$G_k(i + \ell - 1, j) = F(i + \ell - 1, j)$$

$$G_k(i + \ell, j) = (F(i + \ell, j))^*_k$$

(2a) When $F(i + \ell, j) = I$, we have moved across the j^{th} row of R and now must move up one row to row $j + 1$ of F. The idea is to locate the leftmost non-I square of row $j + 1$ and then repeat step 1 for this row.

Suppose our rightmost non-I square in the j^{th} row of F is $(i + \ell, j)$. If $F(i + \ell, j + 1) \neq I$, then create the production: $F_k \to G_k$ where F_k, G_k are grids with values of I except

$$F_k(i + \ell, j) = G_{k-1}(i + \ell, j)$$

$$F_k(i + \ell, j + 1) = F(i + \ell, j + 1)$$

$$G_k(i + \ell, j) = F(i + \ell, j)$$

$$G_k(i + \ell, j + 1) = (F(i + \ell, j + 1))^*_k$$

(2b) When $F(i + \ell, j + 1) = I$, then we must move back across the R squares in the j^{th} row of F until we locate a square with a non-I value in the square above it. To move back across the j^{th} row create the following set of productions: $F_k \to G_k$ where F_k, G_k are grids with values of I except

$$F_k(i + m-1,j) = F(i + m-1,j)$$

$$F_k(i + m,j) = G_{k-1}(i + m,j)$$

$$G_k(i + m-1,j) = (F_k(i + m-1,j))^*_k$$

$$G_k(i + m,j) = F(i + m,j)$$

and $m = \ell, \ell-1, \ldots.$

When $F(i + m, j+1) \neq I$, move up to row $j + 1$ by creating the production in 2a with $\ell = m$.

If we move left over all the R squares in the j^{th} row of F and $F(i + m, j + 1) = I$, then we have "checked" all the squares of R. The last production rule created, $F_k \to G_k$, becomes $F_k \to G$.

(3) If we reach row $j + 1$ of F, we then move left across the squares of R. Suppose we moved up to row $j + 1$ to square $(i+m,j+1)$ of F. If $F(i + m-1,j + 1) \neq I$, create the following set of productions: $F_k \to G_k$ where F_k, G_k are grids with values of I except

$$F_k(i + n-1, j+1) = F(i + n-1, j+1)$$

$$F_k(i + n,j+1) = G_{k-1}(i + n,j+1)$$

$$G_k(i + n-1,j+1) = (F(i + n-1,j+1))^*_k$$

$$G_k(i + n,j+1) = F(i + n,j+1)$$

and $n = m, m-1, \ldots.$

When we have moved left across all of the R squares of row $j + 1$ of F, we return to step 1 and repeat the algorithm.

Derive a set of production rules as above for every production in P, and let P' be the new set of productions. Let N^* be the set of auxiliary symbols we have created, and let $N' = N \cup N^*$. Then

$G_2 = (T, N', S, P')$ is a 5-grammar whose production rules have left members with at most two non-I symbols with at least one of them non-terminal.

The sequence of new productions created to replace a given production can only be used together in the order created. The application of this sequence corresponds to (F,G) in G_1. Therefore, the same scenes are produced, and $L(G_1) = L(G_2)$.

By Theorem 6.2 each 3-grammar is equivalent to a 4-grammar. We have shown that each 4-grammar is equivalent to a 5-grammar. Therefore, by transitivity each 3-grammar is equivalent to a 5-grammar. Q.E.D.

In Chapter V we introduced grid grammars with a brief discussion of the desirability of being able to concisely represent the interrelationships which may exist among the elements of a grid. The example we considered was the analysis of hand-written letters. Any automated process designed to recognize the letters of the alphabet must mimic the mental processes by which the human mind distinguishes one letter of the alphabet from another. Our focus on automata and grammars leads us to a consideration of the following questions. Can we devise a machine which will employ a grammatical approach in the analysis of a picture? Can we find a sequence of productions which lead to a given scene? This would involve being able to identify the right sides of production rules in the scene, replace them with the left sides of the production rules, and in this manner hope to reach the initial grid consisting only of blank symbols.

Parsing a scene, that is, discovering the sequence of productions which created it, must apparently proceed backwards in this trial and

error fashion. There may be several productions which could have been applied as the last production; given each of these, there may have been several which could have been the next-to-last production, etc. If any reverse sequence leads to the initial grid, then the final grid is a scene. We refer to this method as bottom-up parsing. But this process is non-deterministic, and in general there are no bounds on how long the production sequences are or even if they are unique.

Even the one-dimensional case requires severe restrictions on the grammars in order for there to exist a parsing algorithm.

The definition of an automata capable of the parsing required to identify scenes in the language is beyond the scope of this thesis. No satisfactory restriction on grammars exists in the literature which results in a natural class of parser machines. However, one can envision a machine which operates in a non-deterministic fashion by checking all inverse derivations simultaneously. This task could be simplified by assigning probabilities to each production rule.

CHAPTER VII

SUMMARY

A grid is a two-dimensional generalization of a Turing machine tape. It consists of an infinite number of tapes, arranged successively as rows in an infinite array. The machine which operates on a grid is defined in the same way as a Turing machine. It consists of a control unit, a read/write head, and an unbounded array of squares used for input and output. All but a finite number of the squares are initially blank. A grid machine involves alphabet and state symbol sets and a computing function which determines the behavior of the machine on a given grid. The behavior of a grid machine differs from that of a Turing machine by the ability of the read/write head to move in four directions -- left, right, up, down. A grid machine can perform as a computational device, a function evaluator, and an acceptor automata.

The squares of a grid are referenced from an arbitrarily designated square as in the Cartesian coordinate system. The value of an undefined square is denoted by a special symbol I, called an ignore symbol. Grids containing undefined values are referred to as partial grids.

A grid grammar is defined by terminal and non-terminal alphabets, an initial grid, and a set of ordered pairs of partial grids with each pair called a production. The composition of grids is obtained by printing the defined values of one grid into the corresponding squares of the other grid. The application of a production rule utilizes the concept of composition of grids.

The left member of a production rule consists of two types of regions, squares with defined values and undefined squares. These regions are referred to respectively as ignore and non-ignore (non-I) regions. By imposing a set of increasingly severe restrictions on the non-I regions, the following hierarchy of grammars is obtained.

(a) 1-grammar: The non-I region of the left member of each production rule is arbitrary.

(b) 2-grammar: The non-I region of the left member of each production rule is connected.

(c) 3-grammar: The non-I region of the left member of each production rule is connected and finite.

(d) 4-grammar: The non-I region of the left member of each production rule is connected, finite and convex.

(e) 5-grammar: The non-I region of the left member of each production rule is connected and consists of at most two symbols with at least one of them a non-terminal.

(f) 6-grammar: The non-I region of the left member of each production rule consists of a single non-terminal symbol.

It was proved that each 3-grammar is equivalent to a 4-grammar, that each 3-grammar is equivalent to a 5-grammar, and that any 3-grammar is equivalent to a 3-grammar with the non-I region of all left members of the production rules simply connected.

It appears to be difficult to find suitable restrictions on grid grammars which then guarantee that a parsing algorithm exists.

BIBLIOGRAPHY

[1] Chomsky, N. "On Certain Formal Properties of Grammars." Information and Control, vol. 2, 1959. pp. 2, 137-67.

[2] Milgram, D. L., and Rosenfeld, A. "Array Automata and Array Grammars, 2." Tech. Rept. 171, Computer Science Center, Univ. of Md., 1971.

[3] Minsky, Marvin L. Computation: Finite and Infinite Machines. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1967.

[4] Rosenfeld, Azriel. "Some Notes on Finite-State Picture Languages." Information and Control, vol. 31, 1976. pp. 177-184.